2009年度 修士論文

LMNtal実行時処理系SLIMにおける検証機能の最適化

提出日 : 2010年1月28日
指導 : 上田 和紀 教授

早稲田大学大学院基幹理工学研究科
情報理工学専攻
学籍番号：5108B114-3

堀 泰祐

概 要

LMNtal は階層グラフ書換えに基づく並行言語モデルで，並行計算系を含む広範な計算モデルの統合を目指している．LMNtal 実行時処理系 SLIM は，通常のプログラム実行に加え，検証機能として LMNtal をモデル記述言語とする LTL モデル検査と全実行経路を出力する非決定実行機能を持ち，LMNtal による高信頼なプログラム開発をサポートしている．これまでに LMNtal による様々な例題の記述が行われ，LMNtal による検証の有用性が確かめられている．

SLIM の二つ検証機能では頂点をプログラムの状態，枝をルール適用による遷移とする状態空間の構築を行う．状態空間は状態の組み合わせ爆発問題により，小さなプログラムであっても状態数が膨大になり，それに伴い実行時間と使用メモリが増大してしまう問題があった．この問題への対策として，状態遷移系のサイズを劇的に低減する効果のある状態削減手法 Partial Order Reduction が試作され，モデルによっては大幅に状態数を削減できることが確かめられている．

本研究の目的は，SLIM の検証機能でより多くのプログラムを扱えるようにし，有用性を高めることである．そこで，状態空間の構築における所要時間と使用メモリの最適化を行った．また，検証機能の利便性を高めるため，状態数削減をプログラマが明示的に行える機能としてアトミック機能を実装や，初期化ルールの実装を行った．一部の例題で状態のハッシュ値の衝突によって状態の等価性判定の回数の爆発的に増加し，所要時間が増大する問題があったが，膜の正規形表現（canonical representation）を計算し，その結果から新たなハッシュ値を計算することでこの問題に対処した．正規形表現の計算には探索が含まれすべての状態に対して行うと所要時間が増加するため，一定数以上衝突したハッシュ値を持つ状態の膜に対してのみ行うようにした．また，これまで新たな状態を作成する際に状態の膜全体のコピーを行っており全体の所要時間の 10%から 20%ほどを占めていたが，全体をコピーせずルール適用による変更箇所のみを保持する最適化を行った．コンパイラの変更は行わず，各中間命令の処理を修正した．結果として，最大 20%ほど所要時間を削減できた．使用メモリのほとんどは各状態の表現により占められる．そこで，次状態を展開し終えた状態の持つ膜の表現をバイナリストリングにエンコードし保持することによって，若干の所要時間の増加があったが各状態の表現を小さくすることができたできた．エンコードは計算量はプロセス数に対して線形である．状態展開時は DFS で行っているが，DFS スタックが深くなると次状態を展開していない状態数が最終的な状態数の大きな割合になると，展開後にエンコードを行ってもあまり効果はないため，状態生成後に状態のエンコードを行い，必要時にバイナリストリングから状態の復元を行う処理を実装した．結果として全体の使用メモリを 10%から 20%ほどにすることができた．

目 次

図 目 次

表 目 次

第1章　研究の目的と背景

プログラムが鉄道や自動車，航空機といった交通システムから，金融システム，家電や携帯電話と言った組み込み機器など，社会のあらゆる場所で使われるようになった一方で，プログラムの誤動作によって経済的に大きな影響が出たり，人命に関わる事故が起こっている．誤動作の多くはプログラムのバグが原因であり，さらには仕様に誤りがある場合もある．これらのシステムの多くは並行処理を含んでいる．並行処理は逐次処理に比べ，実行時にあり得る状態が遙かに多く，プログラマが正しさを確認するのは用意ではない．こうしたことから，プログラムの正しさを検査する形式手法の必要性が高まっている．

LMNtal[3] は階層グラフ書換えに基づく並行言語である．LMNtal はプロセス・データ・メッセージを統一的に扱うことができる計算モデルであり，従来の計算モデル，特に並行計算モデルを統合することを目標としている．

並行言語の特徴と書換え型言語としての豊富な表現力と実用的なプログラミング環境から， LMNtal をモデル記述言語として利用すれば，多くの書換えモデルの表現と検証を容易に行うことができる．また，専用のモデル記述言語を必要とせずに開発と検証を同一の言語で行うことができる．そこで，LMNtal 実行時処理系 SLIM は，通常のプログラム実行に加え，検証機能として LMNtal をモデル記述言語とする LTL モデル検査と全実行経路を出力する非決定実行機能を持ち，LMNtal による高信頼なプログラム開発をサポートしている．これまでに LMNtal による様々な例題の記述が行われ，LMNtal による検証の有用性が確かめられている．

SLIM の二つ検証機能では頂点をプログラムの状態，枝をルール適用による遷移とする状態空間の構築を行う．状態空間は状態の組み合わせ爆発問題により，小さなプログラムであっても状態数が膨大になり，それに伴い実行時間と使用メモリが増大してしまう問題があった．この問題への対策として，状態遷移系のサイズを劇的に低減する効果のある状態削減手法 Partial Order Reduction が試作され，モデルによっては大幅に状態数を削減できることが確かめられている．

本研究の目的は，SLIM の検証機能でより多くのプログラムを扱えるようにし，有用性を高めることである．そこで，そこで，状態空間の構築における所要時間と使用メモリの最適化を行った．また，検証機能の利便性を高めるため，状態数削減をプログラマが明示的に行える機能としてアトミック機能を実装や，初期化ルールの実装を行った．

1.1 論文構成

本論文の構成は下記の通りである.

第 2 章 LMNtal 言語モデルについて述べる.

第 3 章 LMNtal の処理系について述べる.

第 4 章 LMNtal 実行時処理系 SLIM について述べる.

第 5 章 SLIM における検証機能の最適化について述べる.

第 6 章 実験結果について述べる.

第 7 章 まとめと今後の課題について述べる.

第2章　LMNtal

本章では，言語モデル LMNtal について説明する．本章の内容は文献 [3, 5] に基づいている．

2.1　概要

LMNtal は階層グラフの書換えに基づく言語モデルであり，計算もデルとしての簡潔性とプログラミング言語としての実用性の両立を目指している．LMNtal の基本データ構造は，階層グラフでありアトムを基本要素として，それを膜とリンクの二つの手段で構造化したものである．膜は多重集合を構成し，多重集合は入れ子にできる．また，リンクはアトム同士を一対一で接続する．どちらの構造も動的再構成が可能である．

2.2　LMNtalの基本構成要素

LMNtal の基本構成要素はアトム，リンク，膜，ルールである．

アトム　（m 価）のアトムはアトム名と m 個（$m \geq 0$）の順序づけられた引数からなる．それぞれの引数は（自分自身または他の）アトムの引数につながるリンクの端点である．また，名前と価数の組をファンクタと呼ぶ．

リンク　リンクはリンク名を用いて表記し，同じリンク名を持つアトムの引数同士が相互接続されていることを表す．LMNtal では後述の構文条件（リンク条件）を設けて，アトムとリンクの両者が（ハイパーグラフではない）無向グラフ構造を表す．

膜　膜は他の要素（膜を含む）を囲うことで階層グラフを構成する．この階層グラフ構造を LMNtal ではプロセスと呼ぶ．また，膜の階層構造の親子関係において，ある膜の親，子，兄弟，子孫の関係にある膜をそれぞれ親膜，子膜，兄弟膜，子孫膜と呼ぶ．また，あるプロセスを含むもっとも内側の膜をそのプロセスの所属膜という．また，また，リンクによってたどることのできるアトムと膜の集合を分子と呼ぶ．

P ::=	$\mathbf{0}$	(空)
	$p(X_1, \ldots, X_m)$ $(m \geq 0)$	(アトム)
	P, P	(分子)
	$\{P\}$	(セル)
	T :- T	(ルール)
T ::=	$\mathbf{0}$	(空)
	$p(X_1, \ldots, X_m)$ $(m \geq 0)$	(アトム)
	T, T	(分子)
	$\{T\}$	(セル)
	T :- T	(ルール)
	@p	(ルール文脈)
	\$$p$[$X_1, \ldots, X_m$ | A] $(m \geq 0)$	(プロセス文脈)
	$p(*X_1, \ldots, *X_m)$ $(m > 0)$	(アトム集団)
A ::=	[]	(空)
	$*X$	(リンク束)

図 2.1: LMNtal の構文

ルール　プロセスの書換え規則をルールと呼ぶ．ルール *Head* :- *Guard* | *Body* の左辺（Head, Guard）は書き換えるべきプロセスのテンプレートと条件であり，右辺（Body）は書換え後のプロセスのテンプレートである．膜はルールを 0 個以上保持することができ，その多重集合をルールセットと呼ぶ．

2.3 構文

2.3.1 基本構文と拡張構文

LMNtal の基本構文を手短に説明する．詳細については [3] を参照されたい．LMNtal の構文は図 2.1 のように定義される．X_i は**リンク名**，p は**アトム名**であり，P は**プロセス**である．具象構文ではリンクは大文字から始まる識別子，アトム名は小文字から始まる識別子で表現する．T はプロセスの書換え規則の表現に用いる**プロセステンプレート**であり，局所文脈 (特定のセルの内部での文脈) を扱う機能をもつ．

0 は中身のないプロセス，$p(X_1, \ldots, X_m)$ は m 価アトム，P, P はプロセスの並列合成，$\{P\}$ は膜 { } によってグループ化されたプロセス，T :- T はプロセスの書換え規則である．

LMNtal におけるプロセスは，**同じリンク名が 2 回を超えて出現してはならない**

という**リンク条件**を満たさなければならない．あるリンク名の各出現はそのリンクの端点を表し，それらの集合がリンクを表す．

プロセス P に 1 回だけ出現するリンク名は P の**自由リンク** (P の外部につながるリンク) を表し，P に出現するそれ以外のリンク名は P の**局所リンク**を表す．個々のルールの各リンク名は，そのルール内にちょうど 2 回出現しなければならない．

ルールを膜に入れることができ，膜は計算の局所化のために利用できる．ルールは，そのルールが所属する膜やその子孫膜の内容を書換えることができるが，親膜の内容を書換えることはできない．

ルール文脈は膜の中のすべてのルールの多重集合とマッチし，**プロセス文脈**は膜の中のルール以外のプロセスのうち，明示的に指定されていないもの全体とマッチする．個々のルール中の文脈の出現はいくつかの構文条件を満たさなければならない [3]．アトム，ルール文脈，プロセス文脈は，同じ名前 p を持っていても互いに無関係である．

プロセス文脈の引数は，自由リンクの出現に関する制約条件を指定するものである．ルール左辺のプロセス文脈 `$`p`[`$X_1, \ldots, X_m$ `|` A`]` の引数 $X_1, \ldots, X_m$ は，その文脈が持っていなければならない自由リンクを指定しており，これをプロセス文脈の**明示的な自由リンク**と言う．

2.3.2 基本型と拡張構文

LMNtal では，整数と浮動小数点数を，`8(X)` ，`3.14(Y)` のように，その数値をアトム名とする 1 価アトムで表現する．数値アトムの引数はその数値を参照するプロセスにつながる．LMNtal では数値もプロセスであり，数値型をはじめとする基本型 (組込みの型) はプロセス型 (プロセスに対する型)[6] の一種である．

アトムが基本型に属するかどうかの検査や基本型に対する演算を指定するために，LMNtal では，**型付きプロセス文脈**という拡張構文を導入している．通常のプロセス文脈のマッチする相手が膜 (階層構造) によって決定されるのに対し，型付きプロセス文脈のマッチする相手はグラフ構造 (接続構造) とグラフ中のアトム名によって決定される．マッチする相手を指定するために，拡張構文として**ガード**つきルールを用意して，ルールの適用のための付帯条件を指定できるようにしている．

Head `:-` *Guard* `|` *Body*

例 2.3.1 ガードを持つルール

```
a(X), $n[X] :- int($n), $n>0 |
     a(Y), $n[Y], a(Z), $n[Z]
```

は，1 価アトム `a` が正整数アトムにつながっている場合，その構造の複製を作ることを表している．ここでガード条件 `int($n)` は，`$n[X]` が整数アトムを表現する

型付きプロセス文脈であることを求めている．また $n>0 はその整数値が正であることを求めている．$n>0 は同時に $n[X] が整数アトムであることも求めるので，上の例は次のように書いてもよい．

```
a(X), $n[X] :- $n>0 |
     a(Y), $n[Y], a(Z), $n[Z]
```

□

現在の処理系でガード条件として指定できるプロセス型は，unary，int，float，string，ground である．これらはそれぞれ 1 価アトム，整数，浮動小数点数，文字列アトムおよび自由リンクを 1 本だけ持つ無階層グラフを表す．

型付きプロセス文脈はこのように，指定されたプロセス型の構造をもつプロセスにマッチするが，型付きでないプロセス文脈と違い，膜の外に出現することや同じ階層に複数個出現することも許される．

ルール左辺の膜の後に “/” と記述すると，それ以上簡約不能な膜にしかマッチしなくなる．これは，子孫膜における計算終了の検出に利用できる．

ルールのヘッドの前に “$name$ @@” と書くことでルールに $name$ という名前がつけられる．この名前はプログラムの挙動には影響を与えないが，実行時の出力情報に含めることができるのでデバッグや実行時の挙動の把握に使える．

膜にも名前 (膜名) をつけることができる．膜名は単なる注釈ではなく，ルールのヘッドでは膜は同一の名前を持つ膜にしかマッチしない．$name$ なる名前を膜につけたい場合，“$name$ $\{\ldots\}$” と記述する．

コメントはブロックコメントと一行コメントがあり，それぞれ “/*” と “*/” で囲まれた部分（入れ子不可），行の “%” か “//” 以降がコメントになる．

2.3.3 省略構文

木構造のデータを他言語と同様に記述するための項記法が用意されており，アトム a の第 k 引数として，(リンク名のかわりに) 最終引数を省略したアトム b を書くと，a の第 k 引数と b の最終引数とがリンク接続されているものと見なす．たとえば $f(g(x))$ は $f(A), g(B, A), x(B)$ と等しい．A_i を第 i 要素とするリスト構造も $X = [A1, A2, \ldots, An]$ と書ける（X はリストの先頭につながるリンク）．1 価アトム + は前置演算子として使え，’+ ’(X) は $+X$ と書くことができる．

2.3.4 プログラム例

LMNtal のプログラムの例として，LMNtal によるバブルソートを紹介する．

```
L=[X,Y|L2] :- X>Y | L=[Y,X|L2].
```

このルールは，リスト中の任意の隣り合った二つの整数を比較し，前にあった整

数が大きければ順序を逆転させる．整数のリストにこのルールを適用すると，プログラムの終了時（ルールが適用できなくなった状態）にはリストが昇順にソートされる．例えば，

```
l=[0,5,3,4,2,1].
L=[X,Y|L2] :- X>Y | L=[Y,X|L2].
```

を実行すると，結果のリストは

```
l=[0,1,2,3,4,5].
```

となる．

第3章　JavaによるLMNtal処理系

本章では，Java による LMNtal 処理系について説明する．本章の内容は [5] に基づいている．

3.1　概要

Java による LMNtal 処理系は，実行時処理系とコンパイラを中核に持ち，さらに様々な機能を備えている．

開発目的と指針は以下である．

- LMNtal プログラム実行環境を早期に提供する
- 実装研究から言語設計へのフィードバックを行う
- 階層グラフ書き変えの非同期実行方式を確立する
- 階層グラフ書き変えの基本実行方式を確立する
- 最適化研究の土台を提供する
- 拡張機能の研究開発の土台を提供する

これらのことから，Java による処理系は言語モデルの構文が定めるプロセスの構造や，操作的意味論が定める簡約過程に忠実に設計するこことして，構文や操作的意味論との直接的対応関係を保存しない最適化は行わないとしている．

3.2　処理系の全体像

LMNtal 処理系の流れを図 3.1 に示す．まず，LMNtal ソースをコンパイラによって独自に設計した中間命令列に変換する．中間命令列とは，LMNtal プロセスのマッチング検査や書き換え処理を行う命令の列である．トランスレータは中間命令列を Java ソースファイルに変換し，Java バイトコードにコンパイルし，Java のランタイムにより実行する．解釈実行の場合には実行時処理系が中間命令列を直接解釈実行する．

3.3 コンパイラ

コンパイラは LMNtal ソースプログラムを中間語命令列に変換する．lmntal コマンドに`--compileonly` オプションを渡すことで中間命令列がテキスト形式で出力される．また，`--slimcode` オプションを渡すと，SLIM 用に若干変更された中間命令列が出力される．コンパイル時には最適化オプション（`-O`）を渡すことができ，最適化された命令を出力させることができる．

3.4 実行方式

LMNtal プログラムの実行はルールによる階層グラフ書換えの繰り返しである．階層グラフの初期構造はルールとともにプログラムによって与えられ，ルールの適用が不可能になったら実行を終了する．LMNtal の実行には適用するルールとそのルールが適用するルールの選択の双方に非決定性があり，ルール適用の戦略は処理系に委ねなれている．以下では，あるルールが現在の階層グラフのいずれかの部分に対して適用可能かどうかの検査を**テスト**と呼び，あるルールがどの部分構造に適用できるかの探索を**マッチング**と呼ぶ．テストの方法として**アトム手動テスト**と**膜手動テスト**の 2 種類が実装されている．

アトム手動テストは書き換えられる可能性のあるアトムをアトムスタックによって管理し，取り出したアトムを出発点としてマッチングを行う．膜手動テストは適用できる可能性があるルールを持つ膜を実行膜スタックによって管理し，取り出した膜の各ルールについてテストを行う．両者のテストは相補的な関係にあり，アトム手動テストは高速だが完全性に欠き，膜手動テストは完全性が保証されるがルールの実行に時間がかかる．

3.5 中間命令列

中間命令列について説明する．ルールの適用処理は，マッチングとボディ実行とに分けられる．マッチングはそのルールにマッチするプロセスを探す処理であり，ルールのヘッドとガードに対応する．ボディ実行はマッチしたプロセスを書き換える処理である．命令には必ず成功する命令と，失敗することがある命令がある．ボディ実行に現れる命令は必ず成功し，マッチングに現れる命令は失敗することがある．また，繰り返し命令では以降の命令が失敗するとその命令までバックトラックし，次の処理を行う．

各中間命令は，命令の種類を表す命令番号と，命令の引数のリストからなる．命令の引数は，変数番号、変数番号のリスト、引数位置、ファンクタ、命令列などである．命令列は，その命令列の開始点を指すラベルで表現することもある．ファ

ンクタはアトム名と価数を下線でつないだ形、たとえば2価のアトム a の場合は a_2 という形で記述する．

図3.2は以下のルール，

```
b_to_c @@ a(X), b(X) :- a(Y), c(Y).
```

をコンパイルして生成した中間命令列である．この命令列について簡単に説明する．なおこの命令列は4で述べるSLIMのために若干変更されている．

まず，2行目の--atommatchアトム手動テスト部の始まりを示すが，SLIMではアトム手動テストを行わないために命令は出力されていない．--memmatchは膜手動テスト部の始まりを表す．膜手動テストでは変数番号0に適用するルールの存在する膜が渡される．各命令列の先頭には *spec* 命令が必ずあり，仮引数の数と使用する変数の数を引数に持つ．6行目のfindatomでファンクタa/1を持つ，アトムを見つけ、derefでアトムのリンク先が0引数目であることを確かめ，funcでリンク先がb/1であることを確かめている．commitはルール適用が成功したことを表し，以降の命令は失敗しない．removeatomでマッチしたアトムを膜から削除し，newatomでアトムの生成と膜への追加を行い，newlinkでリンクの接続を行っている．

以下に主要な命令を解説する．なお，以下の説明で確認するというのは確認できなかった場合には命令が失敗することを表す．

spec [***formals***, ***locals***] 仮引数 (*formals*) と局所変数 *locals* の個数を宣言する．

findatom [***dstatom***, ***srcmem***, ***funcref***] 膜 *srcmem* にあるファンクタ *funcref* を持つアトムを一つ取得し，*dstatom* に設定する．後続の命令が失敗するとこの命令まで制御が戻り，同じファンクタを持つ他のアトムを取得し，再度後続の命令を実行する．

deref [***dstatom***, ***srcatom***, ***srcpos***, ***dstpos***] アトム *srcatom* の第 *srcpos* 引数のリンク先が第 *dstpos* 引数に接続していることを確認したら，リンク先のアトムへの参照を *dstatom* に代入する．

func [***srcatom***, ***funcref***] アトム *srcatom* がファンクタ *funcref* を持つことを確認する．

commit [***ruleref***] ルールのマッチングに成功したことを表す．*ruleref* はルール名である．

removeatom [***srcatom***, ***srcmem***, ***funcref***] 膜 *srcmem* にあってファンクタ *funcref* アトム *srcatom* を *srcmem* から取り出す．

newatom [***dstatom***, ***srcmem***, ***funcref***] 膜 *srcmem* にファンクタ *funcref* を持つ新しいアトム作成し，参照を *dstatom* に代入する．

newlink [***atom1***, ***pos1***, ***atom2***, ***pos2***, ***mem1***] アトム *atom1*（膜 *mem1* にある）の第 *pos1* 引数と，アトム *atom2* の第 *pos2* 引数の間に両方向リンクを張る．

freeatom [***srcatom***] アトム *srcatom* のリソースを解放する．

movecells [***dstmem***, ***srcmem***] （親膜を持たない）膜 *srcmem* にある全てのアトムと子膜を膜 *dstmem* に移動する．

relink [***atom1***, ***pos1***, ***atom2***, ***pos2***, ***mem***] アトム *atom1*（膜 *mem* にある）の第 *pos1* 引数と，アトム *atom2* の第 *pos*2 引数のリンク先（膜 *mem* にある）の引数を接続する．

unify [***atom1***, ***pos1***, ***atom2***, ***pos2***, ***mem***] アトム *atom1* の第 *pos1* 引数のリンク先（膜 *mem* にある）の引数と，アトム *atom2*f の第 *pos2* 引数のリンク先（膜 *mem* にある）の引数を接続する．

unifylinks [***link1***, ***link2***, ***mem***] リンク *link1* の指すアトム引数とリンク *link2* の指すアトム引数との間に双方向のリンクを張る．

copycells [***dstmap***, ***dstmem***, ***srcmem***] 膜 *srcmem* の内容のコピーを作成し，膜 *dstmem* に入れる．

dropmem [***srcmem***] 膜 *srcmem* を破棄する．

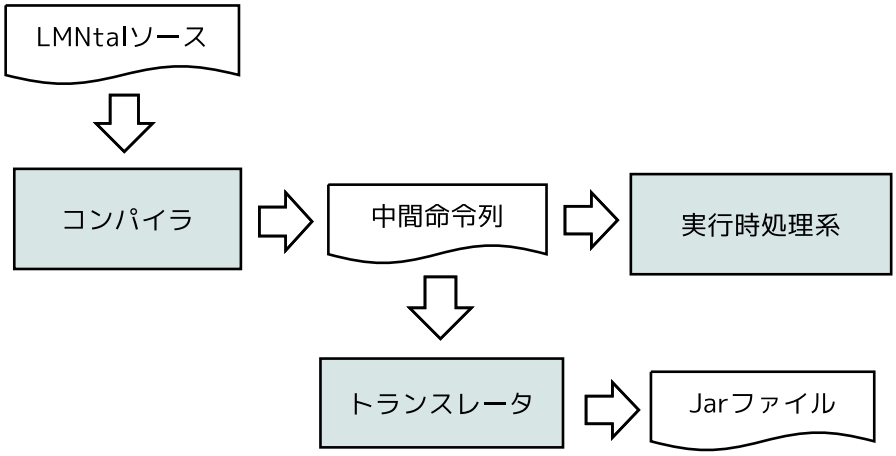

図 3.1: 処理系の流れ

```
 1: Compiled Rule
 2:     --atommatch:
 3:         spec           [2, 5]
 4:     --memmatch:
 5:         spec           [1, 5]
 6:         findatom    [1, 0, 'a'_1]
 7:         deref       [2, 1, 0, 0]
 8:         func           [2, 'b'_1]
 9:         commit         ["b_to_c", 0]
10:         dequeueatom    [1]
11:         dequeueatom    [2]
12:         removeatom     [1, 0, 'a'_1]
13:         removeatom     [2, 0, 'b'_1]
14:         newatom     [3, 0, 'a'_1]
15:         newatom     [4, 0, 'c'_1]
16:         newlink        [3, 0, 4, 0, 0]
17:         enqueueatom    [4]
18:         enqueueatom    [3]
19:         freeatom       [1]
20:         freeatom       [2]
21:         proceed        []
```

図 3.2: 中間命令列

第4章　LMNtal実行時処理系SLIM

本章ではLMNtal実行時処理系SLIM（Slim Lmntal IMplementation）について説明する．

4.1　概要

SLIMはC言語で実装されたLMNtalの実行時処理系である．コンパイラにより生成された中間語命令列を入力として実行を行う．様々な機能を持つJavaによる処理系に比べ，機能は少ないが高速かつ軽量な動作を特徴としている．高速かつ軽量な実行性能のためにわずかな制約があるが，ほとんどすべてのLMNtalプログラムを扱うことができる．

また，通常のプログラム実行だけではなくLMNtalによる高信頼ソフトウェア開発をサポートするための検証機能が実装されている．検証機能には線形時相論理（LTL）による検証属性の記述をサポートするモデル検査機能と，可能な全実行経路を出力する非決定実行機能とがある．

非決定実行機能の出力は頂点をプログラムの状態，枝をルール適用による遷移としたグラフである．この出力によりプログラムの全状態数，ループの有無，最終状態の数などを確認することができ，プログラムの性質の把握や想定外の動作の可能性などを確認することができる．

SLIMは2007年末より開発され，現在では修正BSDライセンスのオープンソースソフトウェアとして開発が行われている．開発の主なメンバーは早稲田大学上田研究室言語班である．

4.2　設計方針

SLIMの開発の目的と方針は以下である．

- 並列実行，組み込み機器での実行，検証ツールとしての土台となる．
- 高速かつ軽量に動作する．
- Javaによる処理系にある機能のなかで，実行性能に影響を与えるものは実装しない．

SLIM の開発開始当時，言語モデルとしての LMNtal は安定し，並列実行や組み込み機器での実行，検証への応用を視野に入れていた．これらの応用を見据えた場合，Java による処理系は実行性能を設計方針としていないため，より実用的な性能を持つ新たな処理系が必要であると結論づけ，SLIM の開発を決定した．SLIM の開発においては，中間語命令の再設計はせずに Java による処理系で生成した中間語命令列を実行する実行時処理系部分の開発のみを行うこととした．中間語命令列の修正が必要となる場合には，Java による処理系に対して SLIM 用の中間語命令列を生成する機能を実装することとした．SLIM の実装言語は高速軽量化を考え C 言語で行うことにした．

Java による処理系には非同期実や分散機能など多くの機能があるが，高速軽量化の妨げになるため，一部の機能を SLIM は実装しないこととした．つまり，SLIM は Java による処理系の実行部分の置き換えではなく，機能は少ないが性能の良いもう一つの実行時処理系とすることにした．

Java による処理系との大きな違いとして，リンクオブジェクトの廃止がある．Java による処理系ではリンクをオブジェクトで表現し，アトムの引数はリンクオブジェクトへの参照を持っている．SLIM ではアトムの引数はリンク先のアトムの参照を直接持つ．また，整数アトムの値をアトムの引数に持たせることで軽量化を図っている．

4.3 実行方式

SLIM ではテストは膜手動テストのみを使う．アトム手動テストを実装しない理由は，ルールのマッチングにおけるプロセスの探索に履歴管理機能 [4] により，膜手動テストのみでも高速な実行が行えると見込んだからである．履歴管理機能ではアトムリストを生成時刻の順に保ち，履歴アトムという特別なアトムをリストの途中に挿入する．履歴アトムより前にあるアトムはマッチング検査が終了しているあとむであり，次のマッチングは履歴アトムより後ろから行う．追加したり書き換えられたアトムはアトムリストの末尾に追加する．

SLIM には履歴管理機能が findatom2 命令として一部が実装されているが，完全性の保証のために findatom2 命令が失敗すると，履歴アトムより前のアトムのマッチングを行っている．これは，履歴管理機能ではアトムのリンク先が変更されると，そのアトムは履歴アトムより後ろに置かれなければならないが，SLIM では実装されていないためである．例えば，

```
a(b(c)). a(c) :- ok.) b(X,Y) :- X=Y.
```

という場合，二つ目のルールが適用されると，アトム b のリンク先 a，c アトムは履歴アトムより後ろに置かれるようにしなければならない．この問題に対処するためにはコンパイラを修正し履歴管理を明示的に行う命令を出力させるか，リン

クを変更する処理にてを加える必要がある．

4.4 プロセスのデータ表現

プロセスを構成するアトム，膜，リンク，ルールの表現は，ルール適用処理の時間性能を悪化させることなく必要最小限のデータのみを持たせるようにした．

4.4.1 アトム

N 引数のアトムは以下のデータを持つ．

- 前のアトムへのポインタ
- 次のアトムへのポインタ
- ファンクタ
- リンク N 個

前のアトムへのポインタと次のアトムへのポインタはアトムを双方向リスト（アトムリスト）で管理するために用いる．ファンクタはアトムの種類を表す．Java による処理系ではアトムは所属膜への参照を持っていたが，SLIM ではアトム手動テストを行わないために必要なくなった．

ファンクタは 2 バイト，リンクは 1 バイトのリンク属性と，1 ワードのリンクデータで表す．N 引数のアトムのワード数は以下の式となる．W は 1 ワードあたりのバイト数である．

$$2 + \lceil (2 + N) / W \rceil + N \tag{4.1}$$

4.4.2 リンク

リンクの端点（アトムの引数）はリンク属性とリンクデータの組で表す．Java による処理系とは異なり，リンクのオブジェクトは存在しない．リンク属性は 1 バイトのデータで，最上位ビットの値が 0 の場合はリンク先が通常のアトムで，残りの 7 ビットにはリンク先のアトムの引数番号である。最上位ビットの値が 1 の場合はリンク先が特別なアトムであることを表し，残りの 7 ビットでアトムの種類を表す．特別なアトムの種類は現在，整数アトム，浮動小数点数アトム，スペシャルアトムがある．リンクデータはリンク先のアトムへの参照だが，リンク先が整数アトムなら整数の値，浮動小数点数なら浮動小数点数への参照となる．

4.4.3 膜

膜を表現するためのデータについて説明する．膜は階層構造を，親膜への参照，兄弟膜を双方向リストで管理するための前後の兄弟膜への参照，子膜のリストの先頭への参照により構成する．アトムはファンクタごとに管理する．同じファンクタのアトムでアトムリストを作り、膜はアトムリストを束ねるハッシュテーブルを持つ．その他に，ルールセットを束ねるベクタ，ルール名を表す整数値，実行終了判定のために stable フラグを持つ．

4.4.4 ルール

ルールの適用処理の表現には 2 種類あり，テキスト表現の中間命令列をバイト列の内部形式に変換したものと，C 言語の関数での表現である．関数は LMNtal のルールでは書けない処理を行う場合や，高速な処理が必要な場合に用いる．

4.5 非決定実行機能

非決定実行機能はプログラムの頂点をプログラムの状態，枝をルール適用による遷移とした状態遷移グラフを構築し表示する機能である．非決定実行機能によりプログラムの全状態数，ループの有無，最終状態の数などを確認することができ，プログラムの性質の把握や想定外の動作の可能性などを確認することができる．

以下のバブルソートを行うプログラムを非決定実行した結果を図 4.1 に示す．

```
l=[3,2,1]. swap @@ L=[X,Y—L2] :- X¿Y — L=[Y,X—L2].
```

States 以下の行は各状態ラベルとプロセスをコロン二つで区切り表示する．Transition 以下の行に各状態ラベルと，次状態の状態番号のラベルを表示する．括弧内は遷移に用いられるルールの名前を（一つ以上）表示する．

4.5.1 非決定実行の流れ

非決定実行機能の流れについて述べる．非決定実行機能では状態とは LMNtal のプロセスのことである．

まず，初期状態として初期プロセスを生成する．状態展開は DFS で行うため，初期プロセスを DFS スタックに積む．DFS では，スタックから状態を *pop* し，状態（プロセス）の全ての膜のルールのマッチングを網羅的に試みる．commit 命令を実行しマッチングの成功がわかると，状態のコピーを作成してコピーした状態をボディ命令で書き換え，次状態を生成する．生成した状態はハッシュ表で実装された状態空間に追加するが，すでに等価な状態が存在した場合にはその状態は

```
States
0::l([3,2,1]). @5
1::l([3,1,2]). @5
2::l([2,3,1]). @5
3::l([2,1,3]). @5
4::l([1,2,3]). @5
5::l([1,3,2]). @5

Transitions
init:0
0::1(swap),2(swap)
1::5(swap)
2::3(swap)
3::4(swap)
4::
5::4(swap)

# of States = 6
```

図 4.1: 非決定実行機能の出力

破棄し，遷移の情報のみを記録する．状態の等価性判定には，状態のハッシュ値と膜の同型性判定を用いる．まず，状態のハッシュ値を計算する．同じハッシュ値を持つ状態がある場合，それらの状態と生成した状態とで膜の同型性判定を行い，すべての状態と同型でなければ状態空間に追加する．同じハッシュ値を持つ状態がなければ同型性判定を行う必要がない．

4.6 モデル検査

SLIM には LMNtal プログラムをを検証の対象とする LTL モデル検査 [2] 機能がある．モデル検査は状態遷移系としてモデル化されたシステムを網羅的に探索することで，そのシステムが要求された性質を満たすか否かを判定する自動検証手法である．

SLIM のモデル検査機能の実装では，探索には nested DFS[2] を用いて，on-the-fly に状態展開を行う．状態展開部分は非決定実行機能と共通した処理を行っている．

第5章　非決定実行機能の最適化

本章では，本研究で実装したSLIMの非決定実行機能の最適化について説明する．

5.1　目的

LMNtalによる様々な例題の記述が行われ，LMNtalによる検証の有用性が確かめられている．

SLIMの検証機能では頂点をプログラムの状態，枝をルール適用による遷移とする状態空間の構築を行う．状態空間は状態の組み合わせ爆発問題 [1] により，小さなプログラムであっても状態数が膨大になり，それに伴い実行時間と使用メモリが増大してしまう問題があった．これまでにこの問題への対策として，状態繊維系のサイズを劇的に低減する効果のある状態削減手法Partial Order Reductionが実装され、モデルによっては大幅に状態数を削減できることが確かめられている。

最適化の研究の目的は、SLIMの検証機能でより多くのプログラムを扱えるようにし，有用性を高めることである．

5.2　設計の概要

本節では，非決定実行機能の最適化の設計について述べる．

5.3　膜とアトムへのIDの付加

5.4　膜の正規化

本節では，膜の正規化について述べる．

ハッシュ値が衝突する状態が多くなると，膜の同型性判定の呼び出し回数が増える．例えば，以下のLMNtalによるバブルソートの例ではハッシュ値の衝突が起こり，膜の同型性判定の回数が多くなっている．

```
l([6,5,4,3,2,1,0]).
sort @@ L=[X,Y |R] :- X>Y | L=[Y,X |R].
```

非決定実行をした結果，状態数が 5040，内部的に生成された状態数 15120 に対してハッシュ値が 7 個となり，その結果膜の同型性判定の呼び出し回数は 2709302 となった．つまり，生成された状態に対して膜の同型性判定が平均 179 回呼ばれている．この例ではリストの要素数を増やすと，呼び出し回数は状態数以上に爆発的に増加する．このように，ハッシュ値の衝突が多い問題では膜の同型性判定の呼び出し回数が増え、所要時間が増加する原因となる．

この膜の同型性判定の呼び出し回数が爆発的に増加する問題への対処法として，膜のハッシュを改良が考えられる．しかし，衝突を減らし，計算時間を増加させず，現在の局所的な構造を抽出するハッシュの方式を改良することは難しかった．そのため，ハッシュ値がある程度衝突した場合に，そのハッシュ値を持つ状態に対して，異なる方式で計算したハッシュ値を使用し状態空間のハッシュテーブルに登録する手法が有効であると考えた．

膜の正規化は，膜をバイナリストリングで表された正規形表現（canonical representation）に変換する機能である．正規形表現を計算することで，その結果から新たなハッシュ値を計算できる．正規形表現から計算したハッシュ値のため，衝突が非常に少ないことが期待できる．

5.4.1 アルゴリズム

膜をバイナリストリングに正規化するアルゴリズムの基本的な方針は，探索によって LMNtal の膜，アトム，ルールを探索により整列し，最小（もしくは最大）となる並べ方を選択することである．例えば，

```
a(X), { b(X, Y), c(Y) }.
```

という構造がある場合，以下の 4 通りの並べ方が存在する．

```
1. a(X), { b(X, Y), c(Y) }.
2. a(X), { c(Y), b(X, Y) }.
3. { b(X, Y), c(Y) }, a(X).
4. { c(Y), b(X, Y) }, a(X),.
```

この中で，辞書順に並べた場合，最小は 1 であり，正規形となる．

正規形表現を計算する手続き id の概略を述べる．

ルート膜では，膜内に含まれる分子の正規形表現を昇順に並べる．

$$id(r\{M_1,\ldots,M_N\}) = \langle MEM\_L\rangle + \langle r\rangle + sort([id(M_1),\ldots,id(M_N)]) + \langle MEM\_R\rangle$$

r 膜名で，は M_I は膜に含まれる分子である．$sort$ はバイナリストリングのリストを整列する手続きである．

分子では，分子に含まれるすべての膜，アトムを基点とした対し正規形表現の中で、最小を選択する．

$$id(P_1, \ldots, P_N\}) = min\,(id(P_1), \ldots, id(P_N))$$

P_I は分子に含まれるプロセスである．$+$ はバイナリストリングを連結する手続きである．min は引数のバイナリストリングの中で最小を選ぶ手続きである．

アトムを基点とする場合，アトムのファンクタと引数のリンク先のプロセスを順に再帰的に連結する．引数のリンク先を変換するためにたどる際には，どのリンクから辿られたかが必要となる．

$$id(a(L_0, \ldots, L_N)) = \langle ATOM \rangle + \langle a/N \rangle + id(linked(L_0), L_0) + \cdots + id(linked(L_N)L_N)$$

link はリンク先のプロセスを表す． アトムのリンク先が親膜のプロセスの場合には，親膜に移動した事を表すタグを置くことで示す．

再帰的にアトムを訪問されたアトムでは，訪問元の引数の位置に，特別なタグを置く．

$$id(a(L_0, \ldots, L_N), \_) = \langle T\_ATOM \rangle + \langle a/N \rangle + id(linked(L_0), L_0) + \cdots + id(linked(L_N), L_N)$$

再帰的に辿られる膜に対しては，辿ってきた自由リンクを含む分子を最初に並べる．

$$\begin{aligned} id(r\{M_1, \ldots, M_k[L], \ldots, M_N\}, L) =& \langle T\_LMEM \rangle + \langle r \rangle + id(M_k) \\ &+ sort([id(M_1), \ldots, id(M_{K-1}), id(M_{K+1}), \ldots id(M_N)]) \\ &+ \langle T\_RMEM \rangle \end{aligned}$$

また，id では訪問したアトムや膜に対して訪問順を訪問番号として記録しており，一度訪問したアトムや膜を再度訪問した位置には訪問番号を置く．

5.4.2 バイナリストリングの形式

正規化表現のバイナリストリングでは，アトムや膜の開始，訪問番号などの情報を 4 ビットタグを置くことで区別している．

5.4.3 非決定実行での利用

非決定実行で一定数以上衝突したハッシュ値を持つ状態は正規形表現を計算している．正規形表現を計算した状態のハッシュ値は正規形表現のバイナリストリング

```
procedure insert_state(s)
 1: state_calc_hash(s)
 2: if state_hash(s) ∈ conflict_hash_values then
 3:    state_calc_mem_id(s)
 4:    if not contains(state_id_tbl, s) then
 5:       insert(state_id_tbl, s)
 6:    end if
 7: else
 8:    if not contains(state_tbl, s) then
 9:       insert(state_tbl, s)
10:       if T ≤ conflict_num(state_hash(s)) then
11:          add_element(conflict_hash_values, state_hash(s)
12:          for all t ∈ states_with_hash(state_tbl, state_hash(s)) do
13:             state_calc_mem_id(t)
14:             insert(state_id_tbl, t)
15:          end for
16:       end if
17:    end if
18: end if
```

図 5.1: 状態の追加処理

から計算し，計算していない状態とは異なるハッシュテーブルで管理する．図 5.1 に状態を状態空間に追加する際の処理を示す．まず，状態 s のハッシュ値を計算し，ハッシュ値が定数 T 以上衝突していた場合は，正規形表現を計算し，ハッシュテーブル *state_id_tbl* に追加を試みる．そうでない場合，ハッシュテーブル *state_tbl* に追加を試み，追加された結果定数 T 以上ハッシュが衝突した場合は，そのハッシュ値を持つ状態すべての正規形表現を計算し，*state_id_tbl* に追加する．

5.5 プロセスのエンコード・デコード

状態爆発により，小さなプログラムであっても状態空間の状態は爆発的に増加し，メモリ容量が不足する問題があった．非決定実行での使用メモリの大部分は状態の表現で占められる．そこで，同じメモリ容量でより多くの状態を表現できるようにするためには，状態の表現をコンパクトにする必要がある．

膜のエンコードは，コンパクトな表現を目的として，LMNtal プロセスをバイナリストリングに変換する．状態空間内の状態はこのバイナリストリングで表現され，必要であればデコードし元の構造を構築する．

5.5.1 アルゴリズム

膜のエンコード処理 *enc_mem* を図 5.2，5.3 に示す．*write* はバイナリストリングにタグやファンクタなどを書き込む手続きである．*visited* はアトムや膜が訪問済みかを調べる手続きであり，*set_visited* 手続きで訪問済みに設定し訪問番号を記録する．*write_visited* 手続きは訪問済みのアトムや膜の訪問番号を書き込む．

エンコードはバックトラックは起こらず計算量は O(プロセス数) である．また，等価な膜であっても同じバイナリストリングが生成されるとは限らない．

5.5.2 非決定実行での利用

非決定実行では状態空間に状態登録する際に膜のエンコードを計算する．状態のプロセスを破棄し，バイナリストリングのみを保持する．状態空間に登録された状態と，登録しようとしている状態でハッシュ値が同じ場合の膜の同型性判定は，エンコードしたバイナリストリングと膜とで行う．

5.6 膜の差分

本節では膜の差分表現について述べる．

非決定実行では状態を展開する際にコピーする処理に手間がかかっていた．非決定実行では状態の膜にルールを適用し，新しい状態を生成する際に，状態の膜をコピーしていた．コピーは生成される状態の数だけ行われる．生成される状態の数は状態空間の遷移数であるが，二つの状態間に複数の遷移がある場合にもそれぞれについて状態は生成される．非決定実行の所用時間のうち，状態のコピーが占める割合は ID 付加による最適化を行った場合は 10%から 20%であり，行わない場合は 50%にまでなることもあった．ルールを適用する前と適用後の膜では変更箇所はプロセスの一部であり，全体をコピーする必要はない．そのため，コピーを行うのではなく，ルール適用による変更情報（差分）のみ作成することで，非決定実行の高速化が行える．

5.6.1 差分の作成

差分の作成は元のプロセスに対してルールのマッチングが成功したとき，つまり，commit 命令を実行した時点で開始し，ボディ命令によるプロセスの書換えを差分として記録していく．差分の情報は各膜ごとに持つものと，グローバルな情報がある．グローバルな情報の中で主なものを以下に示す．

- 新規生成アトム

- 消去されたアトム
- リンクが書換えられたアトム
- 新規生成膜
- 差分情報を持つ膜
- 変更された膜

新規性成アトムや新規生成膜はボディ命令の処理によって新規に生成されたアトムや膜である．差分情報を持つ膜は元のプロセスに含まれていて，アトムや膜，ルールセットの追加や削除といった変更が行われた膜である．子膜が差分情報を持つ膜であっても，親膜もそうであるとは限らない．つまり，膜は元のプロセスに含まれ変更されたか変更されないか，新規作成されたかのの三種類となる．消去される膜は差分情報を持つ必要はなく膜の要素を変更しても問題はないが，現在の実装では差分情報を持たせてしまっている．

各膜の差分情報の中で主なものを以下に示す．

- 追加されたアトム
- 削除されたアトム
- 追加された膜
- 削除された膜
- 追加されたルールセット
- 削除されたルールセット
- 新しい膜名

書き換えを行うボディ命令のうち、アトムや膜の生成・消去，膜へのアトムや膜の追加・削除，リンクの接続，プロセスの移動・コピー・消去を行う主な命令の処理について述べる．基本的にはこれらの処理を行うことでその他の命令の処理も行われる．

アトムや膜の生成・消去 $newatom$，$newmem$，$freeatom$，$free_atom$ 命令ではグローバルな情報に生成や消去記録する．

膜へのアトムや膜の追加・削除 膜にアトムを追加（削除）する場合，膜が新規性成膜であれば膜を直接変更する．差分情報を持つ膜であれば，差分情報の追加されたアトムにアトムを追加（削除）する．膜の場合も同様である．

リンクの接続 *relink*，*unify*，*unifylinks* 命令などではアトムのリンクを書き換える．また，*insertproxies*，*removeproxies* 命令などのプロキシでもアトムのリンクを変更する処理がある．アトムのリンクを変更する際、アトムが新規作成アトムならばそのアトムのリンクを変更するが，そうでなければアトムをコピーして新規作成したアトムに対してリンクの変更を行う．新規作成アトムとコピーされたアトムの対応はグローバルな情報に記録し，以降コピーされたアトムに対する処理は対応する新規生成アトムに対して行う．ちなみに，SLIM ではアトムの再利用を行わないため，*newlink* 命令でリンクを変更するアトムは新規生成アトムとなる．

プロセスの移動・コピー・消去 *copycells* 命令でのプロセスのコピーは，コピー先の膜が新規性成膜であれば直接膜を変更し，差分情報を持つ膜であれば，差分情報としてアトムや膜の追加を記録する．*movecells* 命令でプロセスの移動を行う場合、移動先の膜に対する処理は *copycells* と同じである．移動元の膜は（新規性成膜であることは有り得ない，そのような命令は生成されないはずである），差分情報としてアトムや膜の削除を記録する．また，移動先の膜に新規生成されたプロセスが自由リンクを含む場合、自由リンクが接続されている親膜のプロキシのリンク接続を変更する．*dropmem* 命令でアトムや膜の削除と消去は，差分情報を持つ膜であれば差分情報として記録する．

5.6.2 差分情報の適用と取り消し

差分を使い表現した膜に対してハッシュ値の計算など膜の情報が必要な処理を行う場合には，差分情報を元の構造に破壊的に適用する．破壊的に適用せずに，アトムのリンク先の取得や，膜の持つアトムや膜を得る処理を差分情報を利用するような方式も考えられるが，プロセスの接続構造をたどるプリミティブな処理を変更した場合、処理速度のペナルティが大きいと考えた．差分情報を適用し，必要な処理を終えた後は，適用の逆の処理をして適用を取り消す．この時、元のプロセスのアトムリストの順や兄弟膜の順が変わる可能性があるが，実装では元のプロセスに対するルール適用はすべてべて終えているので問題はない．

図 5.4 に差分情報を適用する処理を示す．*get_mem_delta* は膜の差分情報を，*get_modified* はリンクが書き換えられたアトムに対応する新規生成アトムを取得する手続きである．*commit* 手続きは差分情報を受け取り，膜の差分情報の適用（*commit_mem*）を行い，アトムのリンク接続を修正する．*commit_mem* ではアトム，膜、ルールセット、膜の追加・削除，そして膜名の変更を行う．

図 5.5 に差分情報の取り消し処理を示す．*revert*，*revert_mem* はそれぞれ *commit*，*commit_mem* と反対の処理を行う．

5.7 プロセス辞書

ハッシュ値の計算，膜の同型性判定，正規化，エンコードの処理では膜やアトムをキーとした辞書を，処理済みのアトムや膜の検査や，訪問番号の記録などに頻繁に使用する．しかし，辞書はアトムや膜のオブジェクトのポインタをキーとしたハッシュ表で実装されていたため所要時間の多くを占めていた．そこで，膜とアトムに整数値の ID を付加し，辞書の実装をハッシュ表から ID をキーとする配列に変更した．

ID の付加は膜やアトムの生成時に行う．ID は異なる状態のプロセスであれば重複してもよい．

procedure $enc\_mem(mem)$

1: $set\_visited(mem)$
2: $write(T\_LMEM)$
3: $write(mem\_name(mem))$
4: $enc\_mols(mem)$
5: $enc\_mems(mem)$
6: $enc\_rulesets(mem)$
7: $write(T\_RMEM)$

procedure $enc\_linked\_mem(mem, atom)$

1: **if** $visited(mem)$ **then**
2: $write\_visited(mem)$
3: **else**
4: $set\_visited(mem)$
5: $write(T\_LMEM)$
6: $write(mem\_name(mem))$
7: $enc\_atom(atom)$
8: $enc\_mols(mem)$
9: $enc\_mems(mem)$
10: $enc\_rulesets(mem)$
11: $write(T\_RMEM)$
12: **end if**

procedure $enc\_mols(m)$

1: **for all** $atom \in mem_atoms(m)$ **do**
2: **if not** $visited(atom)$ **then**
3: $enc_atom(atom)$
4: **end if**
5: **end for**

procedure $enc\_mems(mem)$

1: **for all** $m \in children(mem)$ **do**
2: **if not** $visited(m)$ **then**
3: $enc\_mem(m)$
4: **end if**
5: **end for**

図 5.2: 膜のエンコード 1

```
procedure enc_atom(atom)
 1: if visited(atom) then
 2:   write_visited(atom)
 3: else
 4:   set_visited(atom)
 5:   if is_outside_proxy(atom) then
 6:     in_proxy ← get_linked_atom(atom, 0)
 7:     enc_linked_mem(proxy_mem(in_proxy), get_linked_atom(in_proxy, 1))
 8:   end if
 9:   if is_inside_proxy(atom) then
10:     write(T_ESCAPE)
11:     out_proxy ← get_linked_atom(atom, 0)
12:     enc_atom(get_linked_atom(out, 1))
13:   end if
14:   for i = 0 to arity(atom) do
15:     enc_atom(get_linked_atom(atom, i))
16:   end for
17: end if
```

図 5.3: 膜のエンコード 2

procedure *commit*(*root_delta*)

1: **for all** $mem \in$ 差分情報を持つ膜 **do**
2: $commit\_mem(mem, get\_mem\_delta(root\_delta, mem))$
3: **end for**
4: **for all** $src\_atom \in$ リンクが書換えられたアトム **do**
5: $new\_atom \leftarrow get\_modified(root\_delta, src\_atom)$
6: **for** $i = 0$ to $arity(atom) - 1$ **do**
7: **if** new_atom の i 番目のリンク先が new_atom と接続していない **then**
8: new_atom と new_atom の i 番目のリンク先とを接続
9: **end if**
10: **end for**
11: **end for**

procedure *commit_mem*(*mem*, *delta*)

1: **for all** $atom \in$ 追加されたアトム **do**
2: $ush\_atom(mem, atom)$
3: **end for**
4: **for all** $atom \in$ 削除されたアトム **do**
5: $remove\_atom(mem, atom)$
6: **end for**
7: **for all** $m \in$ 追加された膜 **do**
8: $add\_child\_mem(mem, m)$
9: **end for**
10: **for all** $m \in$ 削除された膜 **do**
11: $remove\_child\_mem(mem, m)$
12: **end for**
13: **for all** $r \in$ 削除されたルールセット **do**
14: $remove\_ruleset(mem, r)$
15: **end for**
16: **for all** $r \in$ 追加されたルールセット **do**
17: $add\_ruleset(mem, r)$
18: **end for**
19: $set\_name(mem,$ 新しい膜名$)$

図 5.4: 差分情報の適用

procedure $commit(root\_delta)$

1: **for all** $mem \in$ 差分情報を持つ膜 **do**
2: $revert\_mem(mem, get\_mem\_delta(root\_delta, mem))$
3: **end for**
4: **for all** $src\_atom \in$ リンクが書換えられたアトム **do**
5: $new\_atom \leftarrow get\_modified(root\_delta, src\_atom)$
6: **for** $i = 0$ to $arity(atom) - 1$ **do**
7: **if** src_atom の i 番目のリンク先が src_atom と接続していない **then**
8: src_atom と src_atom の i 番目のリンク先とを接続
9: **end if**
10: **end for**
11: **end for**

procedure $revert\_mem(mem, delta)$

1: **for all** $atom \in$ 削除されたアトム **do**
2: $ush\_atom(mem, atom)$
3: **end for**
4: **for all** $atom \in$ 追加されたアトム **do**
5: $remove\_atom(mem, atom)$
6: **end for**
7: **for all** $m \in$ 削除された膜 **do**
8: $add\_child\_mem(mem, m)$
9: **end for**
10: **for all** $m \in$ 追加された膜 **do**
11: $remove\_child\_mem(mem, m)$
12: **end for**
13: **for all** $r \in$ 追加されたルールセット **do**
14: $remove\_ruleset(mem, r)$
15: **end for**
16: **for all** $r \in$ 削除されたルールセット **do**
17: $add\_ruleset(mem, r)$
18: **end for**
19: $set\_name(mem,$ 元の膜名$)$

図 5.5: 差分情報の取り消し

第6章　最適化の評価実験

前節で述べた各最適化ついて実験により評価を行った．使用した例題は，状態数数千〜数百万程度，実行時間は数秒から 2 千秒程度である．

実験環境を表 6.1 に示す．

OS	Debian 5.0
CPU	Quad-Core AMD Opteron(tm) 2.3GHz, Quad Core × 2(8CPUs)
Memory	16 GB RAM
GCC	4.3.2 -O2 オプションを使用

表 6.1: 実験環境

6.1　メモリ使用量

スタック上の状態を含めて膜のエンコードを行った結果，メモリ使用量は全ての例題で 10%〜30%に削減することができた．

6.2　所要時間

膜の正規化を行った場合，ハッシュの衝突が起こる問題では実行時間の計算量が改善した．膜の正規化以外の最適化を行わなかった場合に対し，すべての最適化を全て使用した場合，速度向上比は図 6.1 となり，最適化前と比べ，2 倍から 3 倍程度の高速化を行うことができた．

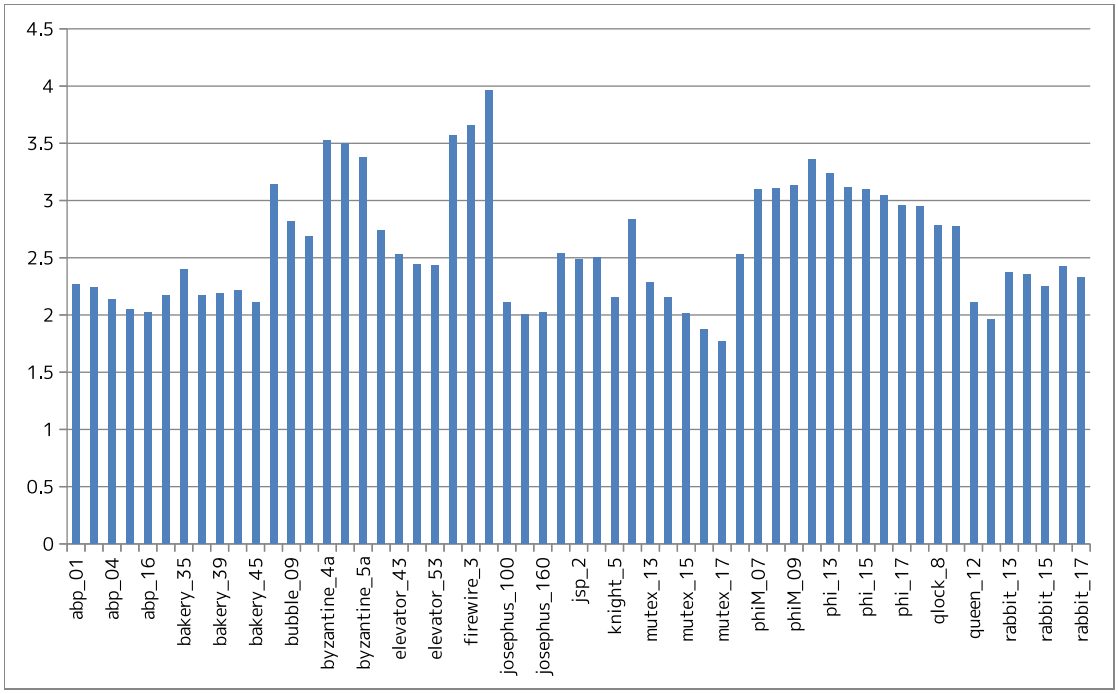

図 6.1: 最適化後の性能向上比

第7章　まとめと今後の課題

本論文では，SLIM における検証機能の最適化手法の設計と実装を行った．膜のエンコードを実装し，メモリ使用量を 10%〜30%ほどに削減することができた．さらなるメモリ使用量の削減手法として，エンコード結果のバイナリストリングの一部の構造を共有や，バイナリストリング自体の圧縮が考えられるが，実行時間の増加を実行時間が増加することが考えられ，今後の課題となっている．膜の正規形表現を実装し，ハッシュ値の衝突による実行時間の増加が起こる問題に対処した．また，膜の差分によりプロセスのコピーの手間をなくし，一部の問題で実行時間が増加したものの，ほとんどの問題で 1 倍から 1.6 倍の高速化を実現できた．差分の情報を利用することでハッシュの計算をインクリメンタルに行うことは今後の課題である．アトムと膜に整数の ID を付加し，アトムと膜をキーにしたの辞書の表現を，ハッシュを使用したものから配列に変更することで，2 倍の高速化を実現できた．最適化の結果，実行時間の多くはルール適用と膜の同型性判定に占められるようになった．この部分を以下に高速化するのかが今後の課題である．

謝辞

本研究や様々な事柄で，多くの方の指導・助言をいただきました．まず，ご指導を賜わった上田 和紀教授に深く感謝致します．上田研の先輩の方々には，研究やその進め方や発表資料の作成について助言をいただきました．特に言語班の先輩にはお世話になりました．心より感謝します．研究設備の管理をして頂いた，admin の方々に感謝します．雑談をし，発表し合い，ともに遊び，学び合ったすべての方に感謝します．

最後に，私に研究という素晴らしい体験をする機会を与えてくれた両親に深く深く，感謝を申し上げます．

2010 年 2 月　堀 泰祐

参考文献

[1] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the state explosion problem in model checking. *Lecture Notes in Computer Science*, Vol. 2000, pp. 176–??, 2001.

[2] G.J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2004.

[3] Kazunori Ueda. Lmntal as a hierarchical logic programming language. *Theor. Comput. Sci.*, Vol. 410, No. 46, pp. 4784–4800, 2009.

[4] 村山敬. 計算量が予測可能な LMNtal システムの設計と実装. 早稲田大学大学院修士論文, 2007.

[5] 村山敬, 工藤晋太郎, 櫻井健, 水野謙, 加藤紀夫, 上田和紀. 階層グラフ書換え言語 LMNtal の処理系. コンピュータソフトウェア, Vol. 25, No. 2, pp. 2_47–2_77, 2008.

[6] 乾敦行, 工藤晋太郎, 原耕司, 水野謙, 加藤紀夫, 上田和紀. 階層グラフ書換えモデルに基づく統合プログラミング言語 LMNtal. コンピュータソフトウェア, Vol. 25, No. 1, pp. 1_124–1_150, 2008.